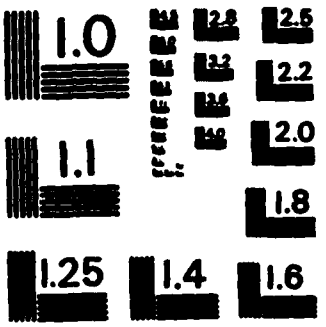
AD-A121 375 FLEX: A WORKING COMPUTER WITH AN ARCHITECTURE BASED ON PROCEDURE VALUES.. (U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) J M FOSTER ET AL.

UNCLASSIFIED JUL 82 RSRE-MEMO-3500 DRIC-BR-85111 F/G 9/2 NL

END

FILMED

DTIC



UNLIMITED

RSRE
MEMORANDUM No. 3500

ROYAL SIGNALS & RADAR ESTABLISHMENT

FLEX : A WORKING COMPUTER WITH AN ARCHITECTURE BASED ON PROCEDURE VALUES

Authors: J M Foster, I F Currie and P W Edwards

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 3500

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3500

TITLE: FLEX: A WORKING COMPUTER WITH AN ARCHITECTURE BASED ON PROCEDURE VALUES

AUTHORS: J N FOSTER, I F CURRIE AND P W EDWARDS

DATE: JULY 1982

SUMMARY

Capabilities need true procedure values to attain their best effect. The Flex architecture, which has been implemented on several machines since 1978, supports both concepts not only in main store but also on (as well as to) backing store and across a network. Capabilities are enforced by tagging them, not by segregating them. This paper describes the architecture and illustrates some of the importance of procedure values, especially for operating systems.

Introduction

The ability to produce, without too much difficulty, machines which can be microcoded has given us new possibilities. First, we can design an architecture and realise it on several hardware configurations. Flex is such an architecture and it has been implemented on three different machines since 1978, with another implementation envisaged (1). Second, we have been able to design a storage allocation scheme, extending to both the main memory and the backing store, in which access is totally controlled and checked by the microcode. Within this we have been able to include an efficient treatment of procedures as true values (2), unlimited by the restrictions of stack based architectures. This has resulted in a substantial increase in flexibility and in the uniformity and sophistication of the control which can be achieved.

The Flex architecture is intended to provide interactive computing for several users. By supporting the use of procedures as true values, aided by capabilities implemented with a tagged memory (rather than by segregation) and extended through the backing store, it provides a particularly safe machine. The instruction code was designed as a target for high level language compilers, particularly in respect of the methods of addressing and the fact that memory allocation and g collection is microcoded. Support for indefinitely many process,

structured values on backing store and the use of procedure values are significant for operating systems. This paper describes the architecture and then indicates how capabilities and procedure values can be combined to give security and to help in the writing of operating systems.

Since Flex is for interactive use, the main processor needs to respond in times comparable with human reactions. By placing the control of peripherals in special peripheral processors we remove from the main processor any need to respond in micro-seconds rather than tenths of a second. Hence a fast microcoded garbage collector (which works in linear time) can be employed, without being detectable by any degradation of performance. On the average about 3% of the time is taken up by garbage collection.

Versions of Flex have been in use for more than three years, and a considerable amount of software exists.

Memory allocation

Data (including code) in Flex is measured in words, bytes or bits and is contained in blocks (called objects in some systems). The programmer does not see a linear store, but an indefinite number of separate blocks each of size between zero and the full capacity of the machine, which he handles by means of pointers which occupy one word. Having a pointer to a block enables one to use only that block and to use it only in a permitted way. The microcode organises the allocation and distibution of these blocks within the real memory. Each word of the real memory has some tag bits which are used to distinguish the words which are pointers from other words and from bits and bytes. The

tag bits cannot be altered by the programmer and are used by the microcode to check the appropriateness of the operations on the word.

Each block has a type which controls what are the legal operations on it. There are six types of which the main ones are :-

1) normal data, which can be read and if the pointer permits can be altered,
2) procedures, which can only be obeyed (with parameters),
3) code, which contains the instructions which are part of the definition of a procedure,
4) workspaces, which are used for the obeying of procedures.

Every instruction checks that its operands are legal. For example, an integer cannot be added to a pointer, nor can access relative to a pointer be used to read or write outside its block. A procedure can be obeyed but neither read nor written and stack operations cannot overflow the block which contains the stack.

Pointers are only created by the microcode and cannot in any way be forged. Hence, although all the users are running together in the same memory, each user can only access the blocks which his current set of pointers gives him, and only in the ways appropriate to each block. Separate or common use of data, or use supervised by means of procedures can be flexibly provided and safely controlled. A pointer to a block of zero size gives a way of creating an unforgeable piece of data which can be used as a key.

In order to represent a reference, that is data which is itself the address of other data, two words are needed - the pointer which says which block is to be used and an offset which says where in the block the data starts. In order to represent a reference to a vector of data

three words are needed - the pointer to the block, the offset of the start of the vector and the number of items in it. The instruction code knows about references and vectors and all operations on them are of course checked for validity. The item to which the reference points and the items which are the components of the vector can be of any size in words, bytes or bits.

The store is garbage collected by the microcode using a fast compacting method which is linear in all its parameters. Because of the speed of the algorithm and the fact that it is obeyed in microcode garbage collection is a small overhead on the operation of the machine.

The backing stores are organised as similar, separate addressing schemes with blocks of similar types containg pointer or non-pointer data and with access controlled in the same way. However, although the main memory can contain pointers into a backing store it is not possible to point from backing store into main memory or from one backing store to another. There is complete freedom to have arbitrarily complex structures on the backing store. We have chosen in practise to have only one alterable block on backing store in order to enforce a regime which is very safe against backing store failures.

The storage scheme serves a purpose similar to normal capabilities, but tagging the pointers instead of segregating them in separate blocks makes it much easier to write compilers, since all the data can be handled uniformly and the parts of what are logically the same structure can be kept together. Also, as will appear, by depending heavily on procedures we obtain a very flexible protection scheme.

The instructions

While a procedure is being obeyed there are three blocks which can be directly addressed by the instructions. These are the local workspace (a special kind of block), the constants block and the non-locals block. The locals block contains the space which is created when the procedure is executed and the constants block contains the data of which the value is known at the time when it was compiled. The function of the non-locals block is described in the next section. A stack is run in the locals block and a register, U, holds the top item of that block. U may contain from zero upwards words, bytes or bits. A typical instruction, such as "load seven bytes from constants starting at offset ten", pushes the contents of U onto the stack in locals, and replaces it by the seven bytes specified, making sure beforehand that the constants block contains the specified bytes and that there is room on the stack to push the old contents of U. Operations such as "plus" act on the contents of U and the items on the stack. An important instruction "push and take n words" pushes U onto the stack and then removes n words from the stack and puts them into U, checking as usual for violations before doing so.

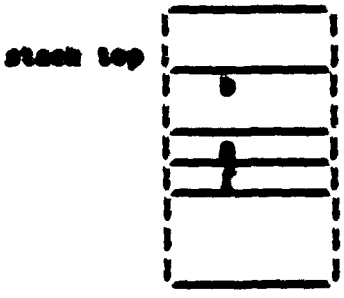
The register U is implemented as follows. If it contains up to two words they are held in registers known to the microcode. Larger values are pointed to by the registers without being moved from where they originated. Ther is no possibility of corrupting the value held in U by some other operation because all operations either use U properly or do not affect the memory at all.

An operation such as "dereference vector of which the components are of size 15 words" expects a reference to a vector (pointer, offset, size of items) in W and loads into W the 15 * size words addressed. Similarly in assignment the contents of W are written into the place specified by the reference or vector on top of the stack or into the local block if that is addressed.

In order to apply a procedure to some parameters a compiler will typically proceed as follows. First the procedure itself (that is a pointer to a procedure block) is loaded, followed by each of the parameters in turn. If f is a procedure of which the value is in the locals block, a is a one word local and b a two word local then we compile f(a, b, 3) into

```
load f size 1        where f is the offset in the local block
load a size 1
load b size 2
load literal 3
```

at which point the situation is like this :-



W holds 3

Figure 1

Then a "push and take" instruction places all the parameters together into W and leaves the procedure at the top of the stack.

```
load f size 1
load a size 1
load b size 2
load literal 3
push and take 4 words
```

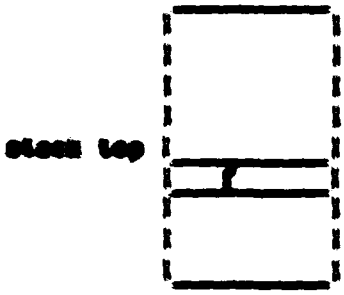


Figure 2

Finally a "call" instruction removes the procedure from the stack and starts executing it using a new locals block and with W holding the parameters.

```
load f size 1
load a size 1
load b size 2
load literal 3
push and take 4 words
call
```

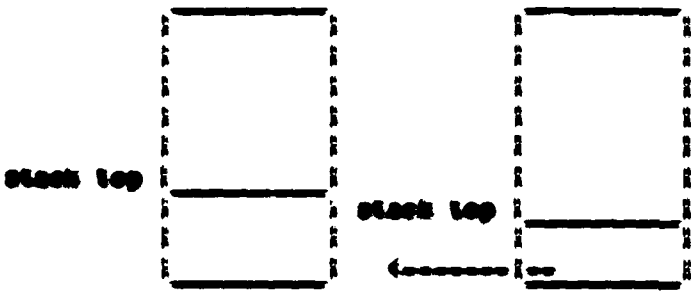


Figure 3

At the end of the new procedure 0 is made to contain all the results and we return to the original procedure and locals block with the results in 0 just as for any other operation.

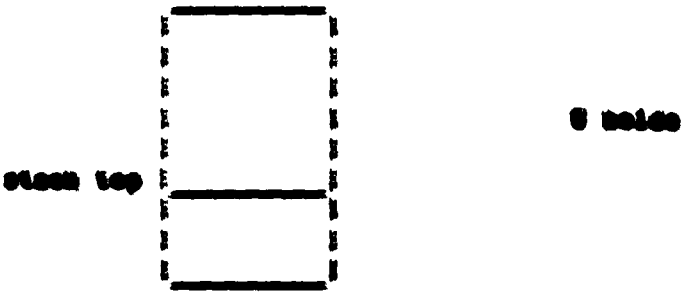


Figure 4

The "call" created a new locals block for the execution of the procedure, though this is optimized in many cases. Those processes,

each of which consists of a sequence of such blocks can be started, stopped and exchange control without any problems about storage, using a mechanism very like a procedure call. Interrupts are treated by calling a procedure value which is special to the interrupt in question, as if from the point in the code where the interrupt occurred.

Unconditional and conditional jumps occur only within the code of a procedure, though there is a mechanism, the "long jump", for going to a continuation specified as a pair consisting of a locals block and a code offset. In this case the locals block must lie on the chain of the current process.

Any illegal operation results in an value in 0 which has type "illegal". An attempt to store such a value will result in the current procedure being left with the same illegal value in 0. There are special instructions to examine illegal values which can be used to determine the error which was made. The programmer can force such errors and the mechanism can be used to provide exception handling.

Procedures

A procedure block contains pointers to blocks which contain the code, the constants and the non_locals. Consider the following piece of program: -

```
...
let i = n+1;
proc f = void: ( ... i ... );
...
```

Whenever we use f we require the use of i in it to give the value which i had at the moment the declaration was encountered (the evaluation of

the declaration). In order to do this in Flex we set up, at the evaluation of the declaration, a non_locals block containing i and all the other items used in f but declared outside it (including references) and we bind this block with the code and the constants to form the procedure. This gives us what is commonly called static binding. Dynamic binding can be provided but is generally undesirable. A pointer to this block now represents the procedure, which can only be executed and cannot be decomposed.

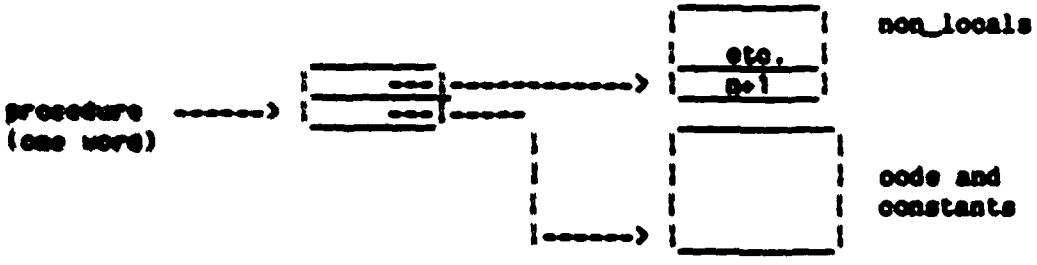


Figure 5

A procedure will protect the non_locals from the garbage collector, but will not enable the user to access the non_locals except in so far as the procedure operates on them. If the declaration was in a loop or a recursion we may get many procedures, each derived from the same code and constants but with different sets of non_locals.

Such procedures are now ordinary data objects, and there is no problem in assigning them into references or in delivering them from procedures.

```
....
proc h = (int j)proc int:
begin ref int t = int := j;
      int:begin t := t + 1; t end
end;
...
proc int k = h(10);
proc int l = h(200);
...
k;  .. k; ... l;...   k;...    l  ...
```

The procedure h takes an integer parameter, j, creates a reference initialised to j and delivers a procedure which, each time it is called, increments the reference and delivers its new contents. The procedures k and l are two such, which will deliver elements of two independent sequences starting with 11 and 201 respectively. Calls of k and l are independent and the fact that we have k gives us no access to the reference which is holding the current number of the sequence. Nor is there any access to the private references from anywhere else in the whole machine (as their would be with capabilities but no procedure values). Any number of such procedures can be set up.

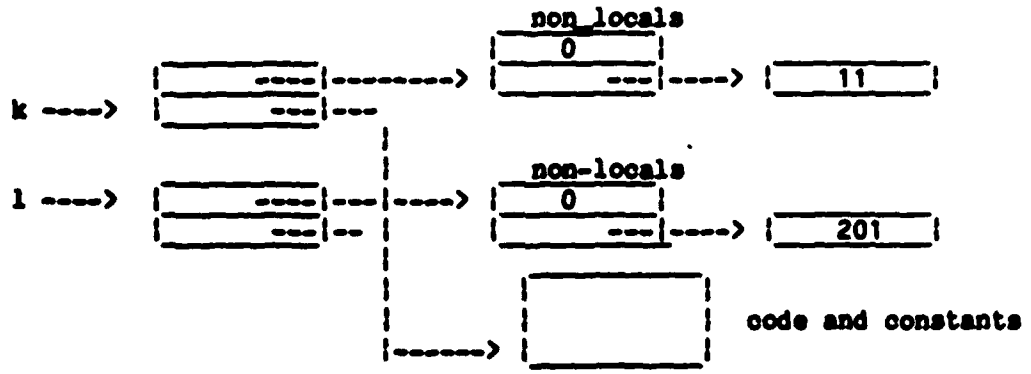


Figure 6

The procedures k and l have just one non-local which is a reference (pointer and offset).

Such procedure values have been known for a long time (2) but have not been widely used because implementing them has involved heavy overheads. But Flex with its low overhead garbage collector can afford to use them freely.

Because they have not been widely used some of their importance has escaped notice, especially in the area of controlling access to data and their relation with capabilities, indeed their necessity if capabilities are to achieve their full potential. In the rest of this paper we concentrate on demonstrating their uses in this area and in the writing of operating systems.

The use of procedure values

Simple policies for controlling the access and use of data are not adequate. Granting permission for such low_level operations as reading and writing on the basis of the identity of the user does not provide a

satisfactory means of control. In general we need to carry out an arbitrary checking action when access to data is mooted, that action depending on the particular data among other information. The use of procedure values is precisely what gives us that ability. If we make the actual data be a non-local of a procedure and give that procedure to a user, he will only be able to see the data or change it by calling the procedure, at which moment the body of the procedure can carry out checks, record footprints or whatever is required. Since the procedure is impenetrable we can be sure that this is all that he can do. Because true procedure values can be created we can set things up so that only the procedure has access to the data, and no-one else including the operating system. Thus we can accomodate various policies and implement unforseen requirements for control.

The operating system is no different from any user in respect of the mechanisms available to afford it protection. It has pointers which give it access to certain blocks which it makes available to the users only through procedures. Indeed the interface between the operating system and a user is precisely through a set of such procedures. A user in his turn can act as an operating system to a sub_user or pass procedures to a parallel user and has just as much sophistication available to him by way of control as has the operating system.

Let us consider an example, a pair of procedures which share a reference to which no one else has access. One procedure writes into the reference, the other reads from it.

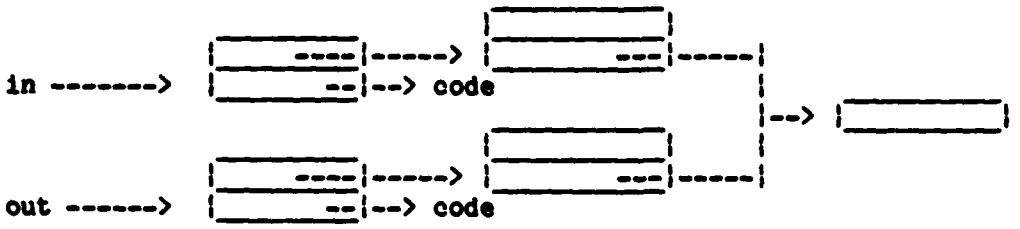


Figure 7

This could be set up by a procedure to generate such pairs of procedures.

```
proc makechannel = struct(proc(int)void in, proc int out):
begin ref int i = int;
      ( (int j)void: begin i := j end,
        int: begin i end )
end
```

Any number of calls of makechannel can be made, each of which will set up a new pair of procedures with a new common reference and no one else will have any access to any of the references. Clearly the parameters and bodies of the procedures could be complex and contain any checks or operations on the data. No interaction with the operating system is needed to set up the scheme. A scheme in which the references were generated by a program which might still have access to them would be undesirable.

Another typical use of procedure values is shown in the way in which the operating system gives a user procedures to manipulate the display on his vdu. The basic code for displaying is bound together with the pointers owned by the operating system which identify the vdu

to form a procedure which will only affect the particular vdu in question. This procedure is then given to the user. A similar technique can be used for output from files, where the data is bound with code to form a procedure which, each time it is called, gives the next line of the file. This is used not only to provide control, but also so that programs which use the lines do not need to know how the file is represented, but only need the specification of the procedure.

Backing store procedures can be created which have a similar nature to those in memory. Once again they consist of code and constants (on the backing store) which can be bound to a set of non-locals (on the backing store) to form a backing store procedure. The only operation available on this is to bring it into memory and convert it into a normal procedure. This facility can be used for many purposes. For example, the operating system creates dictionaries which consist of a number of procedures, including one to look up an identifier and find the corresponding value and one to insert a pair consisting of an identifier and a value. These have bound into them the backing store data structures which actually represent the information in the dictionary but this is totally inaccessible except through the procedures. Thus the integrity of the data can be assured. The user is given the procedures and can only do his manipulations through them. If he wishes he can create in terms of these procedures others, either in memory or on backing store, which offer a subset of the facilities, check accesses or whatever he requires and pass these to other users or to his sub_users. Furthermore this again helps to preserve a constant interface, since it is the specification of the procedures which has to

be kept constant rather than the actual data structure representing the dictionaries.

In fact only a small kernel of procedures is supplied initially and operating systems are built up on top of these without needing any special facilities to do so.

References

1. I. F. Currie, J. M. Foster, P. W. Edwards. Flex firmware. Royal Signals and Radar Est. Report No. 81009

2. P. J. Landin. The mechanical evaluation of expressions. Comput. J. Vol. 6, No. 4, p308 - 320 Jan. 1964)